

# 第六章 子程序结构

- ◆ 子程序又称为过程
  - 它相当于高级语言中的过程和函数
- ◆ 为什么需要子程序
  - 程序段共享
  - 模块化设计
  - 简化程序设计
  - 节省存储空间
- ◆ 使用子程序的主要优点
  - 节省存储空间
  - 减少程序设计时间

# 本章的主要内容

- 6. 1 子程序的设计方法
- 6. 2 嵌套与递归子程序
- 6. 3 子程序举例
- 6. 4 DOS系统功能调用

## 6.1 子程序的设计方法

### 6.1.1 过程（子程序）定义

- 定义语句是伪操作，只告诉汇编程序如何处理，生成合适的机器指令

- 格式： `procedure name PROC Attribute`

.....

`procedure name ENDP`

- 过程名（procedure name）是子程序入口的符号地址
- 类型属性（Attribute）： **NEAR**、 **FAR**

## ◆ 过程属性的确定原则：

- NEAR属性：调用程序和子程序在同一代码段中（段内调用）
- FAR属性：调用程序和子程序不在同一代码段中（段间调用）

## ◆ 80X86汇编程序在汇编时用过程属性确定CALL和RET指令属性

- 用户只需在定义过程时考虑属性，CALL和RET指令属性可以不考虑让汇编程序确定

code segment

main proc far

.....

call subr1

.....

ret

main endp

subr1 proc near

.....

ret

subr1 endp

code ends

segx segment

subt proc far

.....

ret

subt endp

.....

call subt

.....

segx ends

segx segment

.....

call subt

.....

segx ends

## 例6.1 调用程序和子程序在同一代码段中

code segment

main proc far

.....

call subr1

.....

ret

main endp

subr1 proc near

.....

ret

subr1 endp

code ends

保存返回地址  
IP；确定目标  
指令的IP

只恢复  
返回地址的IP

code segment

main proc far

.....

call subr1

.....

ret

subr1 proc near

.....

ret

subr1 endp

main endp

code ends

过程定义  
可以嵌套，  
即一个过  
程定义中  
包含多个  
过程定义

主过程应  
定义为  
FAR属性。  
它是DOS  
调用的一个子过程

NEAR  
属性

## 例6.2 调用程序和子程序不在同一代码段中

FAR属性

恢复返回地址的IP, CS

segx segment

subt proc far

.....

ret

subt endp

.....

call subt

.....

segx ends

segx segment

.....

call subt

.....

segx ends

FAR属性子程序可以被同一段内或不同段内程序调用，而NEAR属性子程序只能被同一段内程序调用

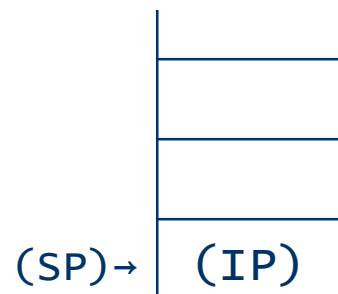
保存返回地址CS, IP;  
确定目标指令的CS, IP

## 6.1.2 子程序调用和返回

子程序调用：隐含使用堆栈保存返回地址

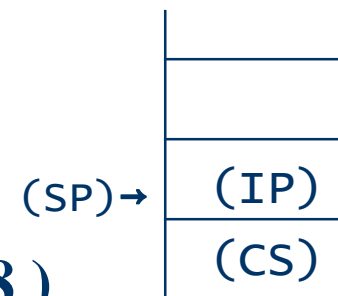
**call near ptr subp**

- (1) 保存返回地址
- (2) 转子程序



**call far ptr subp**

- (1) 保存返回地址
- (2) 转子程序



子程序返回：ret (或者 ret imm8 )



## 6.1.3 保存和恢复寄存器

- 为什么子程序中要保存和恢复寄存器
  - 子程序是独立的共享模块，对寄存器使用具有独立性，这样会产生主程序和子程序使用寄存器冲突
  - 为了解决主程序和子程序使用寄存器冲突，保证主程序正确运行，子程序中必须保存相关使用的寄存器

◆ (1) 保护和恢复寄存器的方法

- 子程序开始时，使用**PUSH**指令保存
- 子程序返回前，使用**POP**指令恢复
- 保存和恢复次序应该相反

子程序设计时应特别注意正确使用堆栈，及堆栈状态变化。一般情况下，子程序中**PUSH**和**POP**指令必须配对使用！

```
subt   proc   far

        push   ax
        push   bx
        push   cx
        push   dx
        .....
        .....
        pop    dx
        pop    cx
        pop    bx
        pop    ax

        ret

subt   endp
```

## (2) 确定保护哪些寄存器的原则

- 保护子程序中将要使用的寄存器及标志寄存器即可
  - 子程序独立性强，不了解调用程序的寄存器使用情况
  - 如果了解调用程序的寄存器使用情况，可适量保存
- 用寄存器向主程序回送结果的寄存器不必保存
- **FLAGS**寄存器保存优先，恢复时最后恢复

## 6.1.4 子程序的参数传送

- ◆ 参数传送：调用程序和子程序之间的信息传送
  - 调用时，主程序传送参数给子程序
  - 返回时，子程序返回参数给主程序
- ◆ 参数传送的一般途径
  - 寄存器
  - 存储器

## 参数传送的具体方法：

(1) 通过寄存器传送参数

(2) 通过存储器传送参数

\*子程序和调用程序在同一程序模块中，则子程序可直接访问模块中的变量

\*子程序和调用程序不在同一程序模块中（ 13章 ）

(3) 通过地址表传送参数地址

(4) 通过堆栈传送参数或参数地址

# (1) 通过寄存器传送参数

- 这种传递方式使用方便，适用于参数较少的情况

## 例6.3 十进制到十六进制的转换程序

(从键盘取得一个十进制数,然后把该数以十六进制形式在屏幕显示)

```
decihex segment
        assume cs: decihex
```

```
main    proc far
        push ds
        sub  ax, ax
        push ax
```

```
repeat: call  decibin    ; 从键盘取10进制数，10→2，保存在BX中
        call  crlf       ; 显示回车换行，防止屏幕显示重叠
        call  binihex    ; 2→16，并在屏幕上显示
        call  crlf
        jmp  repeat
        ret
main    endp
```

通过寄存器BX传送参数

## ▪ 从键盘取10进制数，10→2，保存在BX中

```
decibin    proc near
            mov     bx, 0        ; bx初始化
newchar:    mov     ah, 1
            int     21h          ; 从键盘取10进制数键的ASCII码
            sub     al, 30h       ; 0-9的ASCII码30-39
            jl      exit         ; <0退出
            cmp     al, 9d
            jg      exit         ; >9退出
            cbw                ; AL符号扩展到AH
            xchg     ax, bx
            mov     cx, 10d
            mul     cx
            xchg     ax, bx
            add     bx, ax
            jmp     newchar
exit:       ret
decibin    endp
```

10进制数以四位2进制数形式保存在BX中

;返回的10进制数的二进制数在BX中

## • BX中2进制数→16进制数，并在屏幕上显示

```
binihex    proc    near    ; 要显示的二进制数在BX中
            mov     ch, 4
rotate:     mov     cl, 4
            rol     bx, cl
            mov     al, bl
            and     al, 0fh
            add     al, 30h
            cmp     al, 3ah
            jl      printit
            add     al, 7h
printit:    mov     dl, al
            mov     ah, 2
            int     21h
            dec     ch
            jnz     rotate
            ret
binihex    endp
```

16进制数转换成ASCII码

调用DOS功能在屏幕上显示1个字符  
请参看605页附录4约定



## • 显示回车换行

```
crlf      proc      near
          mov      dl, 0dh    ; “回车” 的ASCII码=0dH
          mov      ah, 2
          int      21h
          mov      dl, 0ah    ; “换行” 的ASCII码=0aH
          mov      ah, 2
          int      21h
          ret
crlf      endp
```

```
decihex   ends          ; 程序代码在一个代码段中与前边 “decihex segment ”配对
          end        main ; 程序从main开始执行
```

## (2) 通过存储器直接传送访问参数

- 子程序和调用程序在同一程序模块中，则子程序象主程序一样直接访问数据段中的变量

例6.4 累加数组中的元素

```
data segment
    ary    dw 1,2,3,4,5,6,7,8,9,10
    count  dw 10
    sum     dw ?
```

```
data ends
```

```
code segment
```

```
main proc far
```

```
    assume cs:code, ds:data
```

```
start:
```

```
    push ds
```

```
    sub ax, ax
```

```
    push ax
```

```
    mov ax, data
```

```
    mov ds, ax
```

```
    call near ptr proadd
```

```
    ret
```

```
main endp
```

```
proadd proc near
```

```
    push ax
```

```
    push cx
```

```
    push si
```

```
    lea si, ary
```

```
    mov cx, count
```

```
    xor ax, ax
```

```
next: add ax, [si]
```

```
    add si, 2
```

```
    loop next
```

```
    mov sum, ax
```

```
    pop si
```

```
    pop cx
```

```
    pop ax
```

```
    ret
```

```
proadd endp
```

```
code ends
```

```
end start
```

## 问题：假设数据段定义如下

```
data    segment

ary      dw  1,2,3,4,5,6,7,8,9,10
count    dw  10
sum       dw  ?

ary1     dw  10,20,30,40,50,60,70,80,90
count1    dw  9
sum1     dw  ?

data    ends
```

```
proadd  proc near
        push ax
        push cx
        push si
        lea  si, ary
        mov  cx, count
        xor  ax, ax
next:   add  ax, [si]
        add  si, 2
        loop next
        mov  sum, ax
        pop  si
        pop  cx
        pop  ax
        ret
proadd  endp
```

\* 如果直接访问内存变量，那么累加数组ary和数组ary1中的元素，由于处理的存储单元在子程序中有固定的约定，不能用同一个子程序proadd。多编写几个子程序？

解决办法：

- 1、设置共享的临时参数存放区，调用时主程序先将参数放在临时存放区，子程序处理临时参数存放区中数据，主程序效率不高
- 2、调用时主程序只传送变量地址表给子程序

### (3) 通过地址表传送变量地址

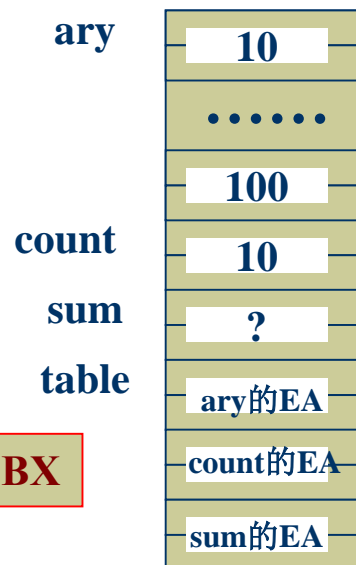
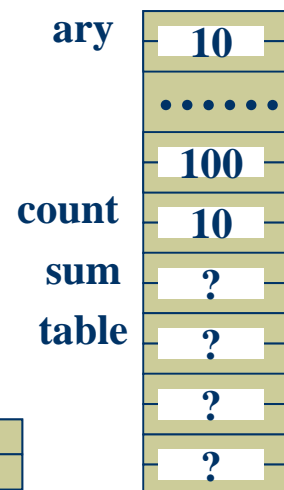
- 适用于参数较多的情况。具体方法是先建立一个地址表，该表由参数地址构成。然后把表的首地址通过寄存器或堆栈传递给子程序

#### 例6.4 累加数组中的元素

```
data segment
    ary    dw 10,20,30,40,50,60,70,80,90,100
    count  dw 10
    sum     dw ?
```

**table dw 3 dup (?)** ; 地址表, 存放ary, count, sum的EA

```
data ends
code segment
main proc far
    assume cs:code, ds:data
start: push ds
       sub ax, ax
       push ax
       mov ax, data
       mov ds, ax
       mov table, offset ary
       mov table+2, offset count
       mov table+4, offset sum
       mov bx, offset table
       call proadd
       ret
main endp
```



**table的EA→BX**

将立即数（变量count的有效地址）送到table+2给出的直接有效地址的两个存储器单元

请解释 `mov table+2, offset count` 的功能

proadd proc near

table的EA在BX中

有效地址

push ax

push cx

push si

push di

mov si, [bx]

mov di, [bx+2]

mov cx, [di]

mov di, [bx+4]

xor ax, ax

next: add ax, [si]

add si, 2

loop next

mov [di], ax

pop di

pop si

pop cx

pop ax

ret

proadd endp

code ends

end start

ary→

10

0000

20

0002

30

40

50

60

70

80

90

100

count→

10

0014

sum →

?

0016

table →

0000

0018

0014

0016

ary

10

.....

100

count

10

sum

?

table

ary的EA

count的EA

sum的EA

bx=0018H 指向table起始地址

si=0000H 指向ary起始地址

di=0014H 指向count单元

xx=10 计数值count

di=0016H 指向sum单元

si指向ary的下一个元素

cx=cx-1; 如cx≠0, 转next

一个框表示2个字节

## (4) 通过堆栈传送变量或变量地址

### ■ 步骤:

1. 主程序把参数或参数地址压入堆栈;
2. 子程序使用堆栈中的参数或通过栈中参数地址取到参数;
3. 子程序返回时使用RET n指令调整SP指针, 以便删除堆栈中已用过的参数, 保持堆栈平衡, 保证程序的正确返回。

### 例6.4 累加数组中的元素

```
data segment
    ary    dw 10,20,30,40,50,60,70,80,90,100
    count  dw 10
    sum     dw ?
data ends
```

```
stack segment
    dw 100 dup (?)
    tos label word
stack ends
```

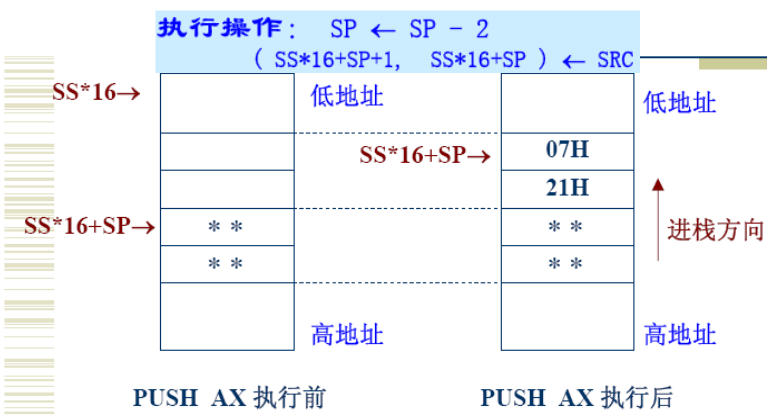
ary→	10	0000
	20	0002
	30	
	40	
	50	
	60	
	70	
	80	
	90	
	100	
count→	10	0014
sum →	?	0016

```
code1 segment
main proc far
    assume cs:code1, ds:data, ss:stack
```

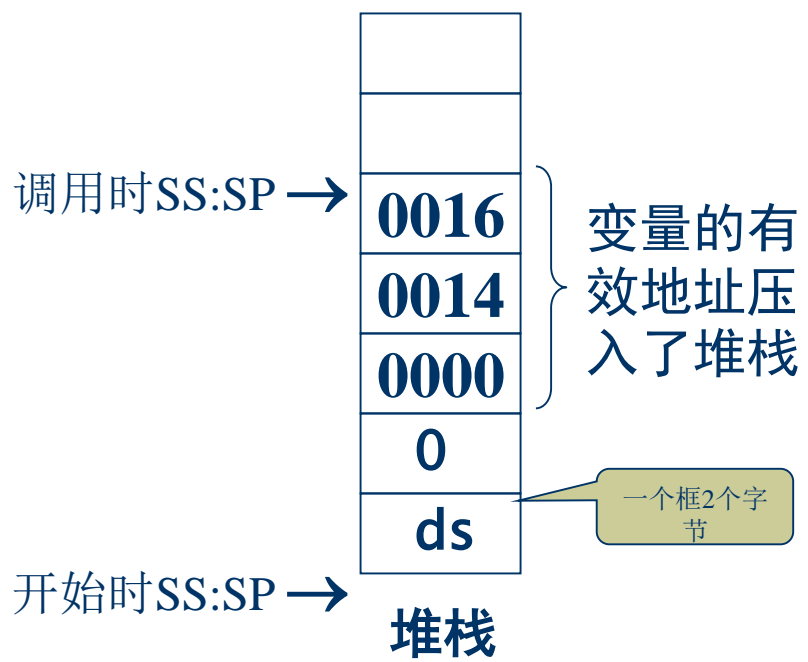
start:

```
    mov ax, stack
    mov ss, ax
    mov sp, offset tos
    push ds
    sub ax, ax
    push ax
    mov ax, data
    mov ds, ax
    mov bx, offset ary
    push bx
    mov bx, offset count
    push bx
    mov bx, offset sum
    push bx
    call far ptr proadd
    ret
```

```
main endp
code1 ends
```



ary→	10	0000
	20	0002
	30	
	40	
	50	
	60	
	70	
	80	
	90	
	100	
count→	10	0014
sum →	?	0016



```
code2 segment
  assume cs: code2
  proadd proc far
```

```
  push bp
  mov bp, sp
```

```
  push ax
  push cx
  push si
  push di
```

```
  mov si, [bp+0ah]
  mov di, [bp+8]
  mov cx, [di]
  mov di, [bp+6]
```

```
  next: xor ax, ax
        add ax, [si]
        add si, 2
        loop next
        mov [di], ax
```

```
  pop di
  pop si
  pop cx
  pop ax
```

```
  pop bp
```

```
  ret 6
proadd endp
code2 ends
end start
```

```
ary→ 10 0000
      20 0002
```

```
count→ 10 0014
sum→   ? 0016
```

sp →

di

si

cx

ax

bp<sub>原</sub>

ip

cs

0016

0014

0000

0

ds

进入子程序后sp  
返回主程序时sp

调用子程序时sp  
不带立即数返回时sp

bp<sub>现</sub> →

bp+6 →

bp+8 →

bp+0a →

ret 6 带立即数返回，返回时将SP修正到 →

一个框表示2  
个字节



■ **结构伪操作STRUC**：定义一种可包含不同类型数据的结构模式，只有具体使用时才有对应存储单元的具体含义

格式：     **结构名 STRUC**

**字段名1   DB   ?**

**字段名2   DW   ?**

**字段名3   DD   ?**

      .....

**结构名 ENDS**

□ **字段名就是变量名，可用变量名表示字段起始地址**

例：学生个人信息

**STUDENT\_DATA   STRUC     ; 4个字段, 18个字节的结构模式**

**NAME       DB    5 DUP (?)**

**ID         DW    0**

**AGE        DB    ?**

**DEP        DB  10 DUP (?)**

**STUDENT\_DATA   ENDS**

- **结构预置语句：**为结构中各字段的数据分配存储器单元，并可为存储单元重新输入字符串和数值

格式1： 变量名 结构名 < >

•采用结构定义中的赋值

格式2： 变量名 结构名 <预赋值说明>

•重新定义结构中的值

等同于数据段中如下定义：

```
S991000.NAME  DB  5 DUP(?)  
S991000.ID    DW  0  
S991000.AGE   DB  ?  
S991000.DEF   DB 10 DUP(?)
```

例： S991000          STUDENT\_DATA      < >

S991001          STUDENT\_DATA      < , 1001, 22, >

STUDENT          STUDENT\_DATA      100 DUP (< >)

- **访问结构数据变量方法：**

```
MOV  AL, S991000.NAME[SI]
```

```
MOV  AL, [BX].NAME[SI]
```

```
STUDENT_DATA STRUC  
NAME  DB  5 DUP(?)  
ID     DW  0  
AGE    DB  ?  
DEF    DB 10 DUP(?)  
STUDENT_DATA ENDS
```

.name可以理解为相对于结构首址的位移量  
bx中存的是结构首址， si给出name字段的第几项  
[BX].NAME[SI]= [BX+.NAME+SI]

## 结构伪操作举例：

### 改写例6.4 累加数组中的元素

```
stack_strc      struc
    save_bp      dw      ?
    save_cs_ip    dw      2 dup (?)
    par3_addr     dw      ?
    par2_addr     dw      ?
    par1_addr     dw      ?
stack_strc      ends
```

定义这个存储数据格式为结构，便于访问编程

sp →

di

si

cx

ax

bp<sub>现</sub> →

bp<sub>原</sub>

进入子程序后sp →

ip

cs

调用子程序时sp →

bp+6 →

0016

bp+8 →

0014

bp+0a →

0000

0

ds

让bp指向结构首址 [bp].save\_bp → bp<sub>现</sub> →

[bp].save\_cs\_ip →

bp<sub>原</sub>

ip

cs

[bp].par3\_addr → bp+6 →

0016

[bp].par2\_addr → bp+8 →

0014

[bp].par1\_addr → bp+0a →

0000

par3\_addr

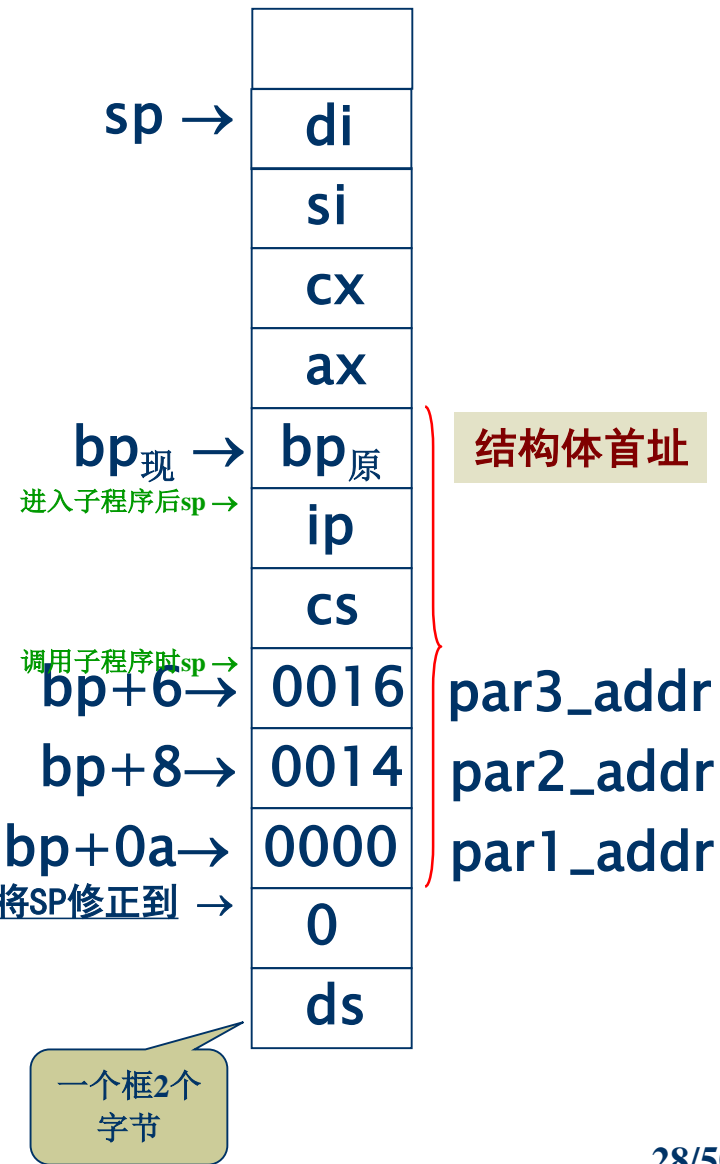
par2\_addr

par1\_addr

```
stack_strc      struc
    save_bp      dw      ?
    save_cs_ip    dw      2 dup (?)
    par3_addr     dw      ?
    par2_addr     dw      ?
    par1_addr     dw      ?
stack_strc      ends
```

```
proadd proc far
    push bp
    mov bp, sp
    push ax
    push cx
    push si
    push di
    mov si, [bp].par1_addr
    mov di, [bp].par2_addr
    mov cx, [di]
    mov di, [bp].par3_addr
    xor ax, ax
next:
    add ax, [si]
    add si, 2
    loop next
    mov [di], ax
    pop di
    pop si
    pop cx
    pop ax
    pop bp
    ret 6
proadd endp
```

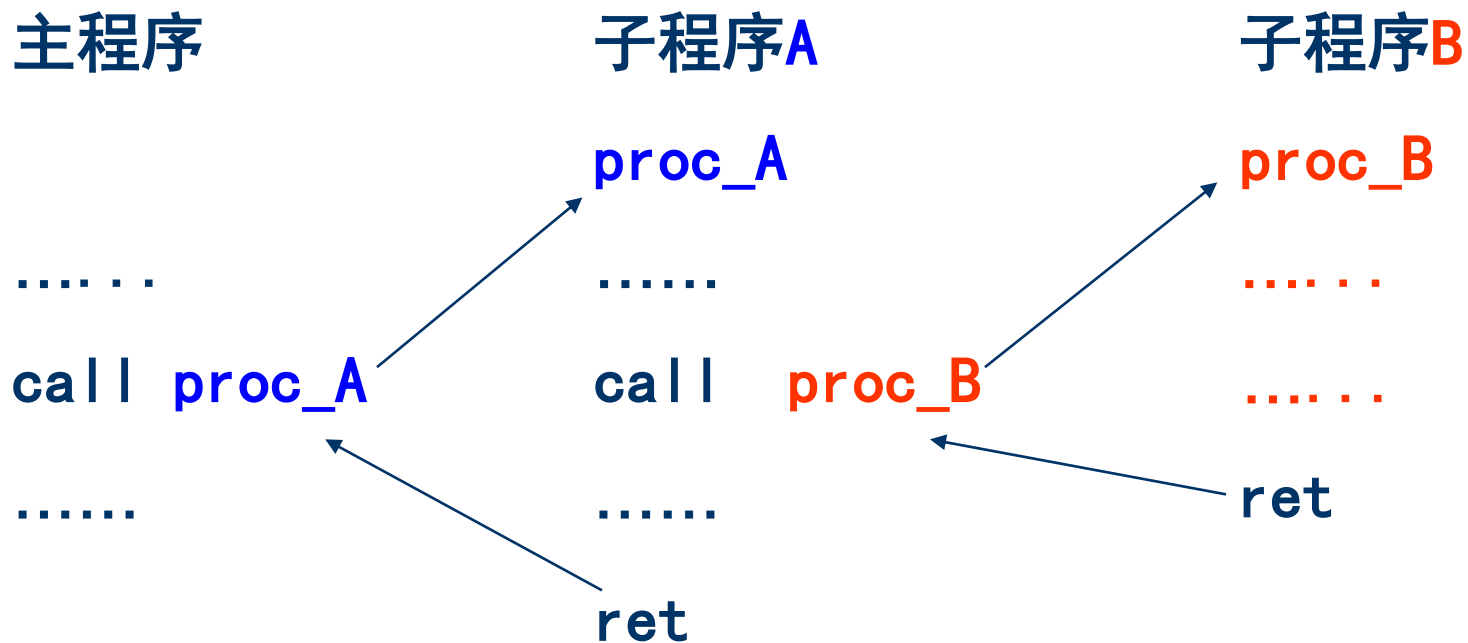
bp指向结构体首址



## 6.2 嵌套与递归子程序

### 6.2.1 子程序的嵌套

■子程序嵌套：一个子程序作为调用程序调用另一个子程序



# 6.2.1 递归子程序

## ◆ 递归子程序

- 递归调用：子程序调用的子程序是它自身
- 递归子程序：递归调用中的子程序
- 是子程序嵌套的特殊情况

但实际中由堆栈大小和现场保护情况决定

## ■ 嵌套深度：是嵌套的层次，层次不限

- 堆栈大小是嵌套深度的关键因素，特别是递归调用
- 特别注意堆栈状态和正确使用

## ■ 注意事项同一般子程序调用

## 例6.7 计算 $N!$ ( $N \geq 0$ )

$$N! = N \times (N-1) \times (N-2) \times \dots \times 1$$

$$\text{递归定义: } \begin{cases} 0! = 1 \\ N! = N \times (N-1)! \quad N > 0 \end{cases}$$

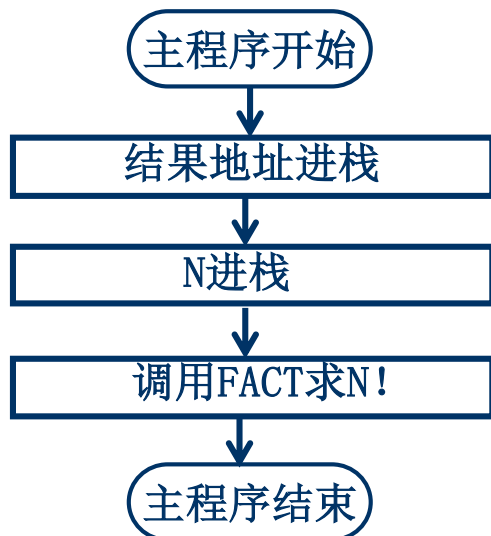
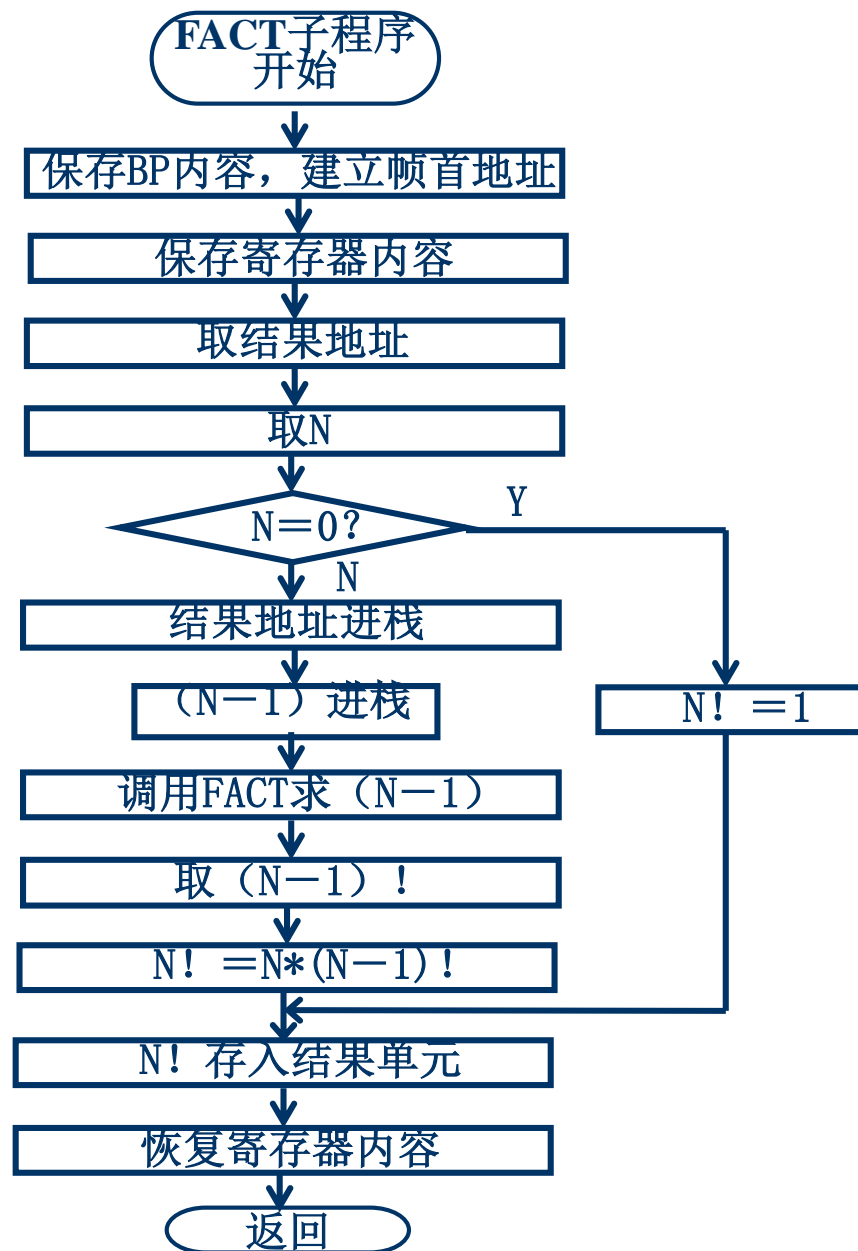


图6.7

$$3! = 3 \times 2!$$

$$2! = 2 \times 1!$$

$$1! = 1 \times 0!$$



## 例6.7 计算n! 假设n=3

```

frame struc
    save_bp      dw    ?
    save_cs_ip    dw    2 dup (?)
    n             dw    ?
    result_addr   dw    ?

```

```

frame ends

```

```

data segment
n_v      dw    3
result    dw    ?
data ends

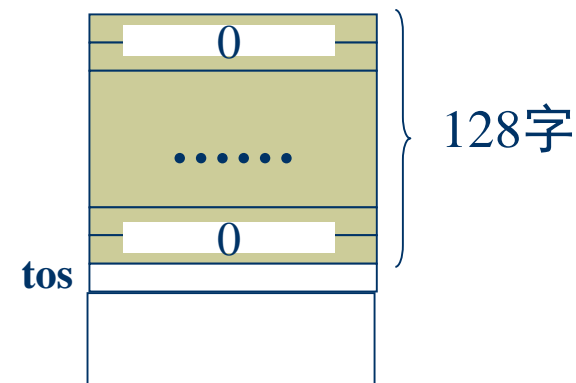
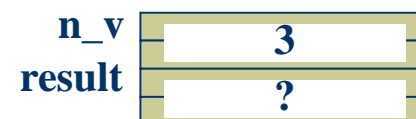
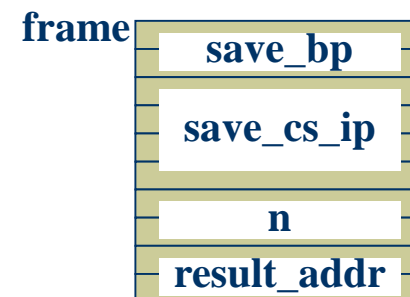
```

```

stack segment
    dw 128 dup (0)
    tos label word
stack ends

```

$3! = 3 * 2!$   
 $2! = 2 * 1!$   
 $1! = 1 * 0!$   
 $0! = 1$





```

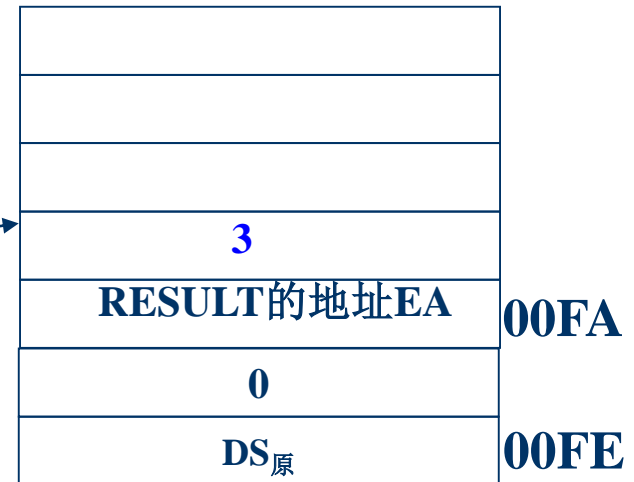
code segment
main proc far
    assume cs:code, ds:data, ss:stack
start:
    mov ax, stack
    mov ss, ax
    mov sp, offset tos
    push ds
    sub ax, ax
    push ax
    mov ax, data
    mov ds, ax
    mov bx, offset result
    push bx
    mov bx, n_v
    push bx
    call far ptr fact
    ret
main endp
code ends

```

```

data segment
n_v    dw    3
result dw    ?
data ends

```



此时sp指向

堆栈

```

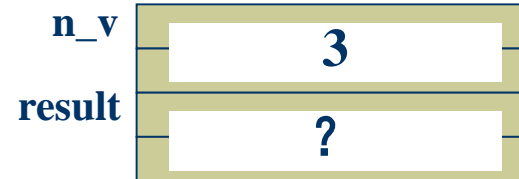
frame struc
    save_bp    dw    ?
    save_cs_ip  dw    2 dup (?)
    n          dw    ?
    result_addr dw    ?
frame ends

```

```

code1 segment
    assume cs:code1
fact proc far
    push bp
    mov bp, sp
    push bx
    push ax
    mov bx, [bp].result_addr
    mov ax, [bp].n
    cmp ax, 0
    je  done
    push bx
    dec ax
    push ax
    call far ptr fact
    mov bx, [bp].result_addr
    mov ax, [bx]
    mul [bp].n
    jmp short return
done: mov ax, 1
return:
    mov [bx], ax
    pop ax
    pop bx
    pop bp
    ret 4
fact endp
code1 ends

```

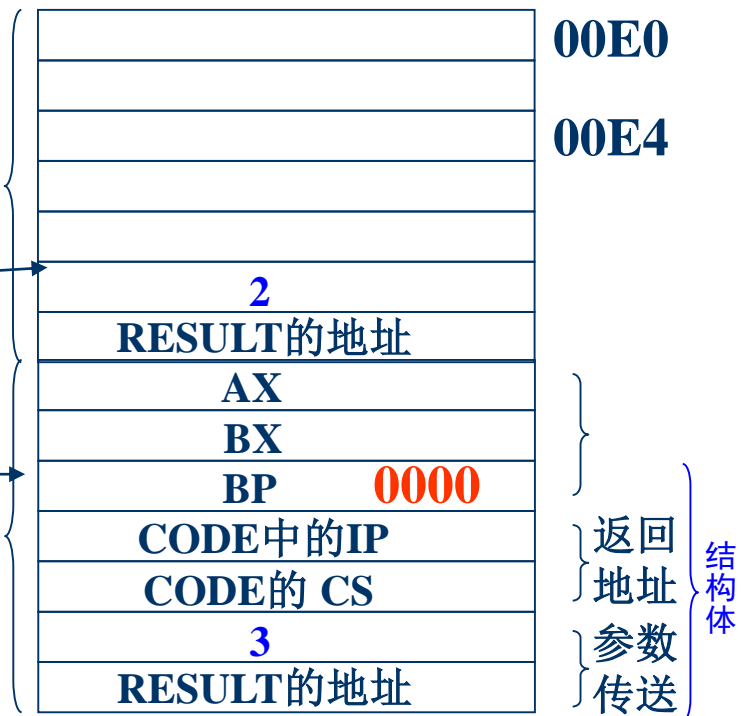


第2帧

此时sp指向

新BP  
第1帧

00FA



```

frame struc
save_bp      dw    ?
save_cs_ip    dw    2 dup (?)
n             dw    ?
result_addr   dw    ?
frame ends

```

n\_v  
result

3
?

```

code1 segment
assume cs:code1
fact proc far
push bp
mov bp, sp
push bx
push ax
mov bx, [bp].result_addr
mov ax, [bp].n
cmp ax, 0
je done
push bx
dec ax
push ax
call far ptr fact
mov bx, [bp].result_addr
mov ax, [bx]
mul [bp].n
jmp short return
done: mov ax, 1
return:
mov [bx], ax
pop ax
pop bx
pop bp
ret 4
fact endp
code1 ends

```

此时sp指向

第3帧

新BP  
第2帧

第1帧

00FA

	00D2
	00D6
1	
RESULT的地址	
AX	00E0
BX	
BP 00F2	00E4
CODE1中的IP	
CODE1的CS	
2	
RESULT的地址	
AX	
BX	
BP 0000	
CODE中的IP	
CODE的CS	
3	
RESULT的地址	

结构体

返回地址  
参数传送

```

frame struc
save_bp    dw    ?
save_cs_ip  dw    2 dup (?)
n           dw    ?
result_addr dw    ?
frame ends

```

```

code1 segment
assume cs:code1
fact proc far
push bp
mov bp, sp
push bx
push ax
mov bx, [bp].result_addr
mov ax, [bp].n
cmp ax, 0
je done
push bx
dec ax
push ax
call far ptr fact
mov bx, [bp].result_addr
mov ax, [bx]
mul [bp].n
jmp short return
done: mov ax, 1
return:
mov [bx], ax
pop ax
pop bx
pop bp
ret 4
fact endp
code1 ends

```

n_v	3
result	?

第4帧

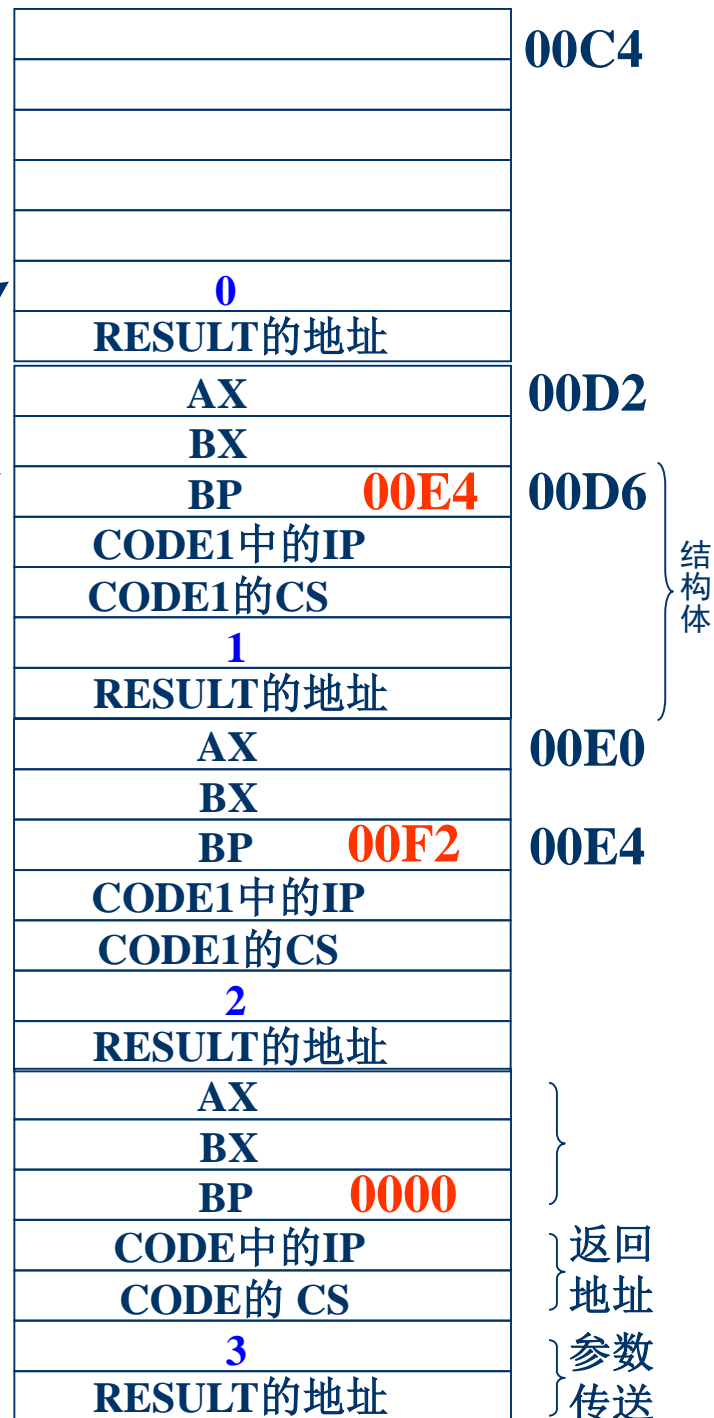
此时sp指向

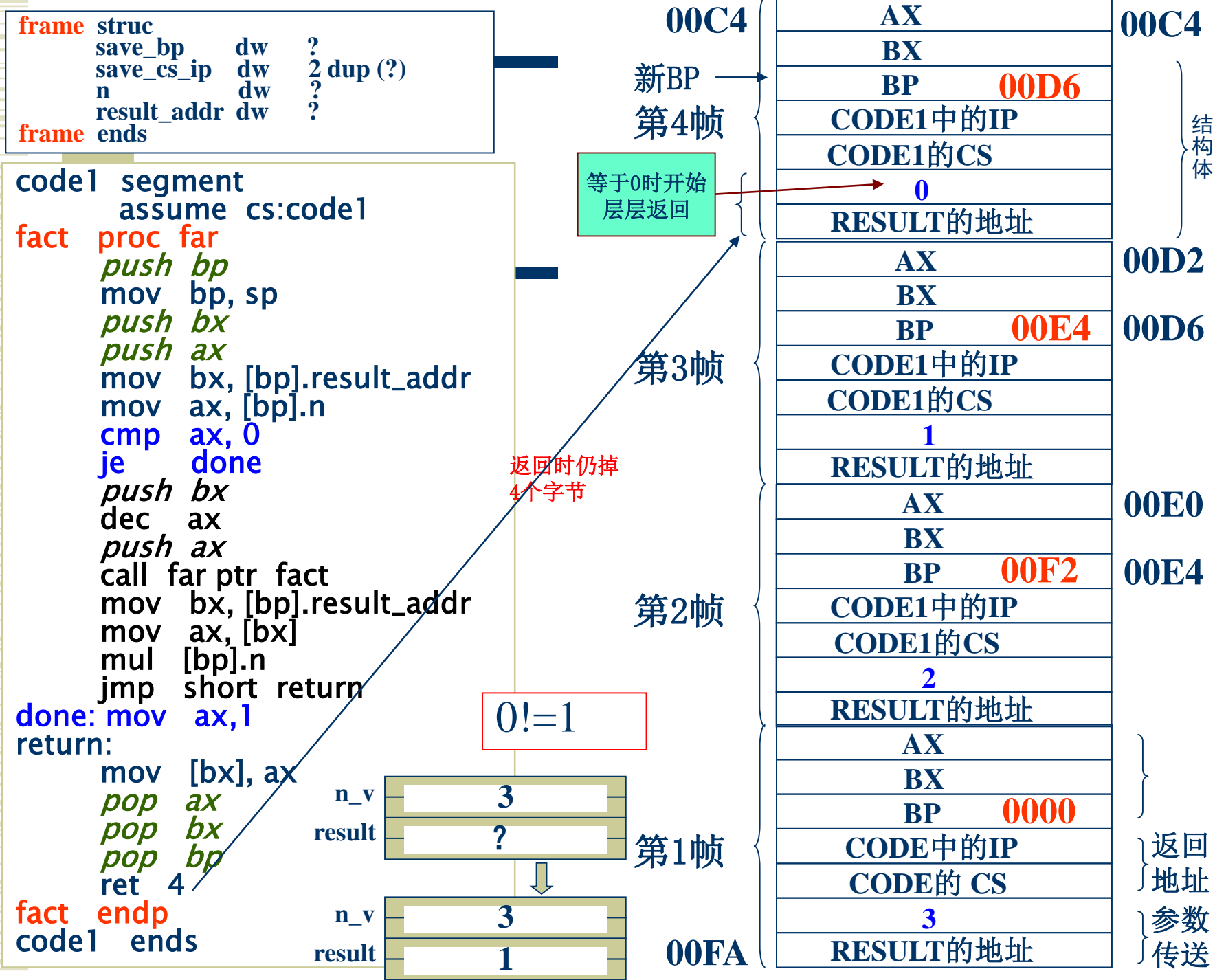
新BP  
第3帧

第2帧

第1帧

00FA





```

frame struc
    save_bp      dw    ?
    save_cs_ip    dw    2 dup (?)
    n             dw    ?
    result_addr   dw    ?
frame ends

```

1!=1\*0!  
0!=1

n_v	3
result	1

↓

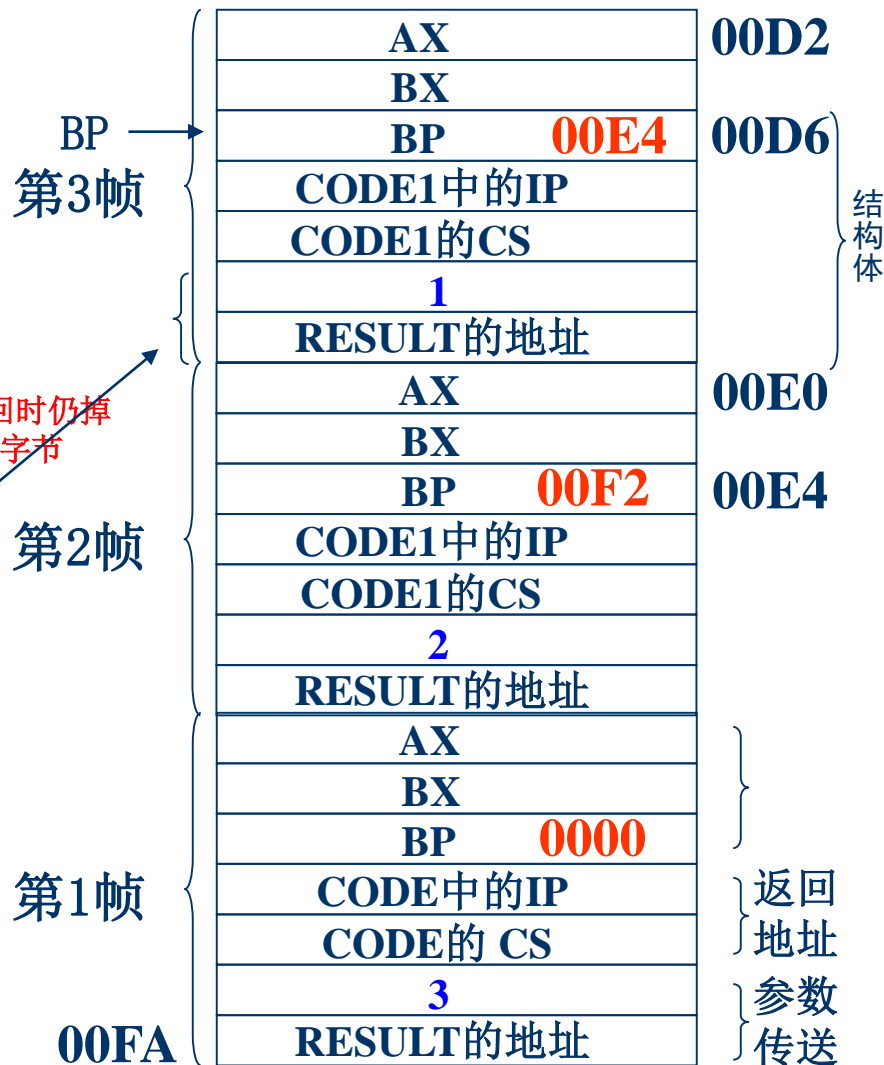
n_v	3
result	1

```

code1 segment
    assume cs:code1
fact proc far
    push bp
    mov bp, sp
    push bx
    push ax
    mov bx, [bp].result_addr
    mov ax, [bp].n
    cmp ax, 0
    je done
    push bx
    dec ax
    push ax
    call far ptr fact
    mov bx, [bp].result_addr
    mov ax, [bx]
    mul [bp].n
    jmp short return
done: mov ax, 1
return:
    mov [bx], ax
    pop ax
    pop bx
    pop bp
    ret 4
fact endp
code1 ends

```

返回时仍掉  
4个字节



```

frame struc
    save_bp    dw    ?
    save_cs_ip  dw    2 dup (?)
    n          dw    ?
    result_addr dw    ?
frame ends

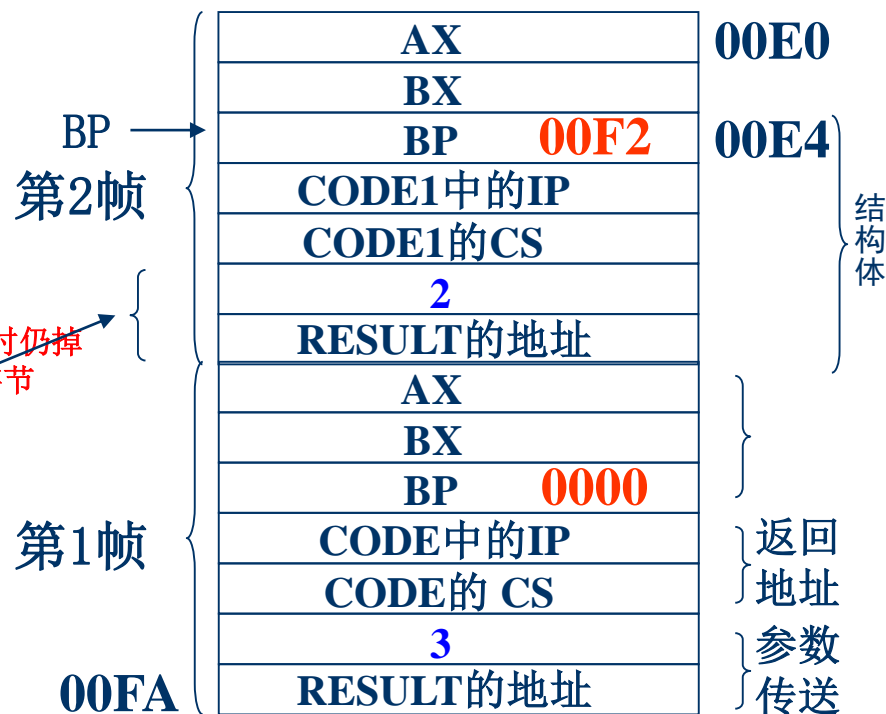
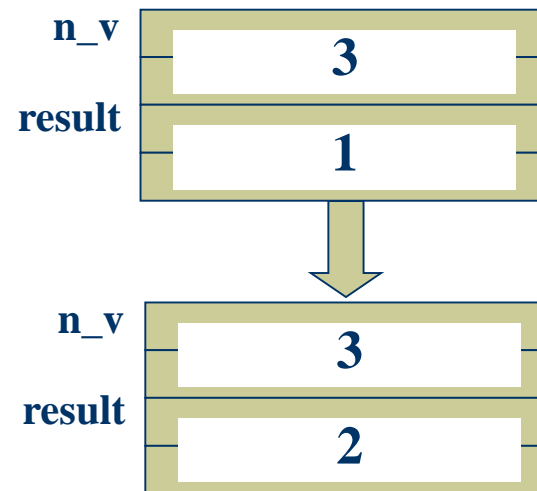
```

$2! = 2 * 1!$   
 $1! = 1 * 0!$   
 $0! = 1$

```

code1 segment
    assume cs:code1
fact proc far
    push bp
    mov bp, sp
    push bx
    push ax
    mov bx, [bp].result_addr
    mov ax, [bp].n
    cmp ax, 0
    je done
    push bx
    dec ax
    push ax
    call far ptr fact
    mov bx, [bp].result_addr
    mov ax, [bx]
    mul [bp].n
    jmp short return
done: mov ax, 1
return:
    mov [bx], ax
    pop ax
    pop bx
    pop bp
    ret 4
fact endp
code1 ends

```



```

frame struc
    save_bp    dw    ?
    save_cs_ip  dw    2 dup (?)
    n          dw    ?
    result_addr dw    ?
frame ends

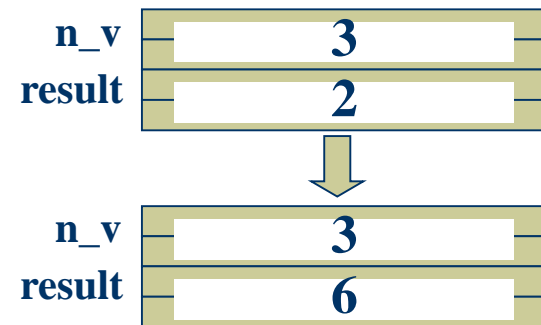
```

```

code1 segment
    assume cs:code1
fact proc far
    push bp
    mov bp, sp
    push bx
    push ax
    mov bx, [bp].result_addr
    mov ax, [bp].n
    cmp ax, 0
    je done
    push bx
    dec ax
    push ax
    call far ptr fact
    mov bx, [bp].result_addr
    mov ax, [bx]
    mul [bp].n
    jmp short return
done: mov ax, 1
return:
    mov [bx], ax
    pop ax
    pop bx
    pop bp
    ret 4
fact endp
code1 ends

```

$3! = 3 * 2!$   
 $2! = 2 * 1!$   
 $1! = 1 * 0!$   
 $0! = 1$



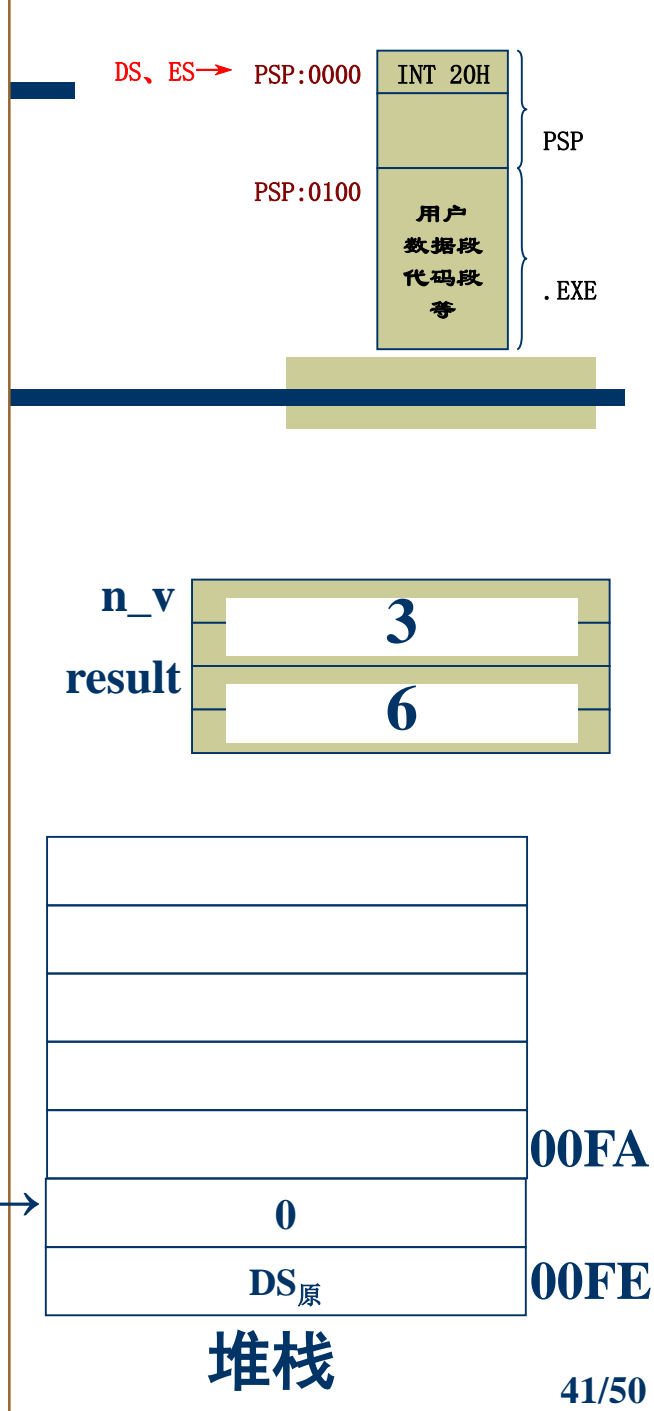
返回时仍掉  
4个字节

00FA



```
code segment
main proc far
    assume cs:code, ds:data, ss:stack
start:
    mov ax, stack
    mov ss, ax
    mov sp, offset tos
    push ds
    sub ax, ax
    push ax
    mov ax, data
    mov ds, ax
    mov bx, offset result
    push bx
    mov bx, n_v
    push bx
    call far ptr fact
    ret
main endp
code ends
```

(源码)



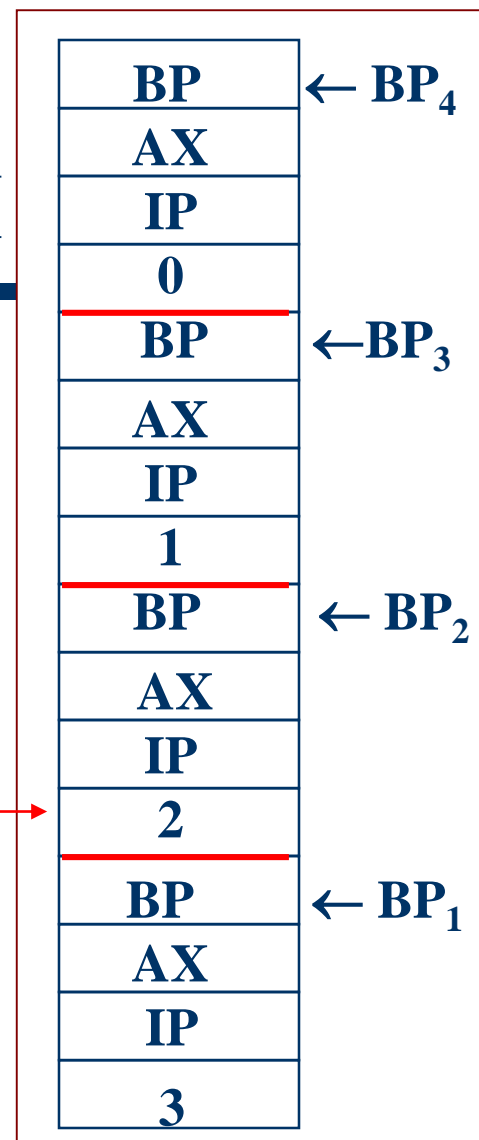
## 例6.7 计算n! 不使用STRUC定义

```
mov bx, n
push bx
call fact
pop result
```

主程序部分

```
fact proc near
    push ax
    push bp
    mov bp, sp
    mov ax, [bp+6]
    cmp ax, 0
    jne fact1
    inc ax
    jmp exit
fact1: dec ax
    push ax
    call fact
    pop ax
    mul word ptr[bp+6]
exit: mov [bp+6], ax
    pop bp
    pop ax
    ret
fact endp
```

传递参数与  
结果共用该单元



使用STRUC定义，结构清晰，  
不易出错，修改方便！

## 6.4 DOS系统功能调用

- ◆ 系统功能调用是DOS为系统程序员及用户提供的一组常用子程序
  - 用户可在程序中调用DOS提供的功能
- ◆ DOS规定用INT 21H中断指令作为进入各功能调用子程序的总入口，再为每个功能调用规定一个功能号，以便进入相应各个子程序的入口。
- ◆ DOS系统功能调用的分类：  
设备管理、文件管理、目录管理

- ◆ **DOS系统功能调用的使用方法（约定）：**
  - 在AH寄存器中存入所要调用功能的功能号；
  - 根据所调用功能的规定设置入口参数；
  - 用INT 21H指令转入DOS系统功能子程序入口；
  - 相应的子程序运行完后,可以按规定取得出口参数

- ◆ **一般调用格式：**
  - ◆ 设置调用参数
  - ◆ MOV AH, 功能号
  - ◆ INT 21H
  - ◆ 取返回参数

```
MOV AH, 1    ; 键盘输入并回显  
INT 21H
```

- ◆ **简单举例：参看P605 附录四——键盘输入单个字符，显示器输出单个字符等**

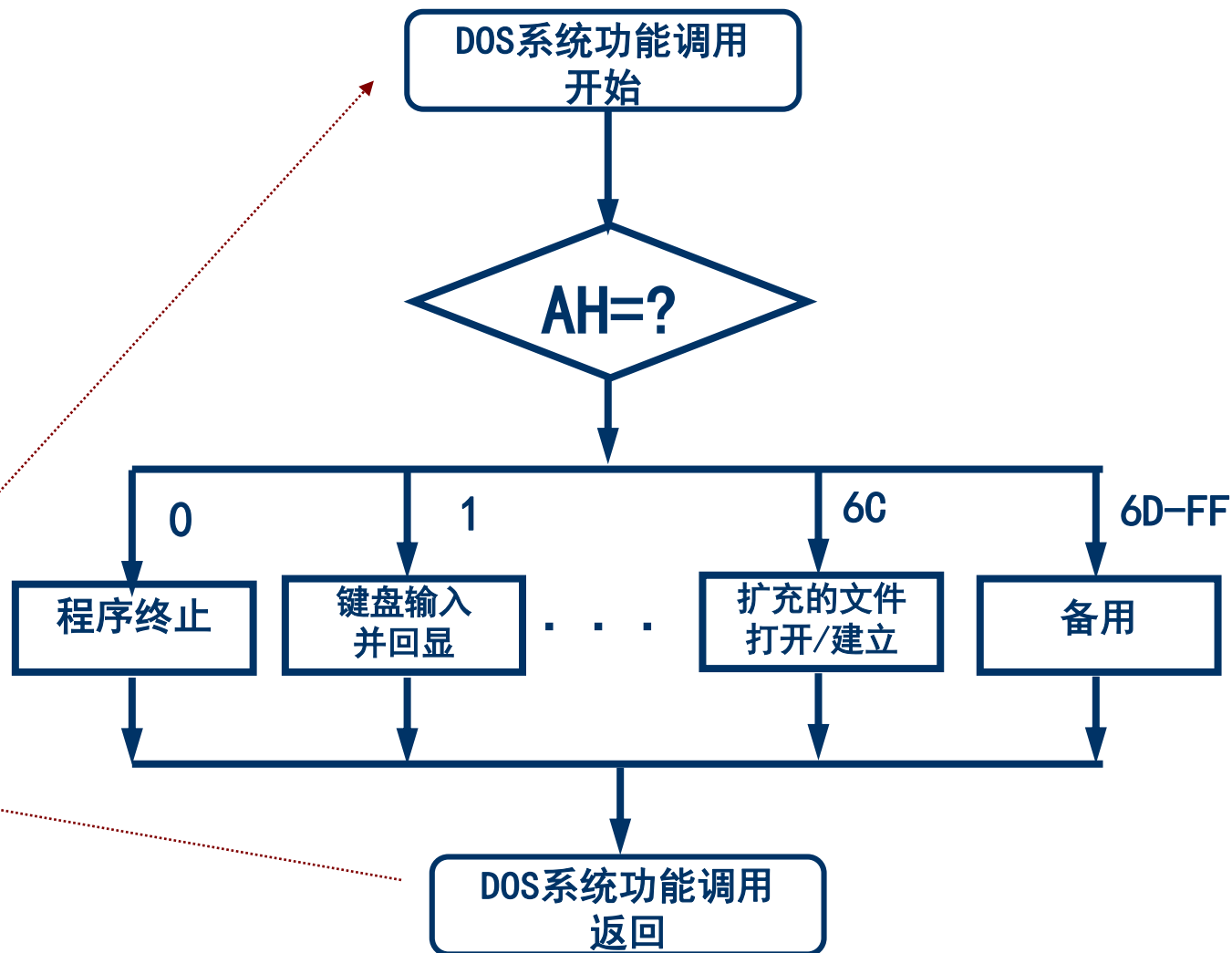
用户程序中：

设置调用参数指令

MOV AH, 功能号

INT 21H

取返回参数指令



多分支结构

## (1) DOS键盘功能调用 (AH=1, 6, 7, 8, A, B, C)

例：单字符输入 (AH=1：键盘输入并回显)

```
get-key:  mov    ah, 1    ; 键盘输入并回显
          int     21h
          cmp     al, 'Y'  ; 键盘输入字符在AL中
          je      yes
          cmp     al, 'N'
          je      no
          jne     get_key

yes:
          .....

no:
          .....
```

例：输入字符串 ( AH=0ah )

定义缓冲区：

方法1    maxlen   db  32  
          actlen   db  ?  
          string   db 32 dup ( ? )

方法2    maxlen   db  32, 0, 32 dup ( ? )

方法3    maxlen   db  32, 33 dup ( ? )

输入字符串    `lea dx, maxlen`  
                 `mov ah, 0ah`  
                 `int 21h`

DS:DX

maxlen→

20

actlen→

0b

string→

‘H’

‘O’

‘W’

20

‘A’

‘R’

‘E’

20

‘Y’

‘O’

‘U’

0d

## (2) DOS显示功能调用 (AH=2, 6, 9)

例：显示单个字符 ( AH=2 )

```
mov  ah, 2  
mov  dl, 'A'  
int  21h
```

例：显示字符串 ( AH=9 )

```
string db 'HELLO', 0dh, 0ah, '$'  
mov  dx, offset string  
mov  ah, 9  
int  21h
```

## (3) DOS打印功能 (AH=5)

例：输出单个字符到打印机 (AH=5)

```
mov  ah, 5  
mov  dl, 'A'  
int  21h
```



## ◆ 设计子程序时应注意的问题

1. 子程序功能定义与说明
2. 参数传递方法
3. 寄存器的保存与恢复
4. 密切注意堆栈状态

# 作业

6.4

6.6

6.9

6.13

初学者如何编写汇编程序？

